

Perl Best Practices

Randal L. Schwartz www.stonehenge.com/merlyn
version 1.5 on 5 May 2006

This document is copyright 2006 by Randal L. Schwartz and licensed under the terms described in
<http://creativecommons.org/licenses/by-nc-sa/2.5/>

What do we mean?

- Damian's PBP has 256 items!
- Not enough time to cover here
 - It would cut into Cinco De Mayo
- Luckily, he selected 30 of the most essential
- Ten each of:
 - Essential Development Practices
 - Essential Coding Practices
 - Essential Module Practices
- We'll start with those

Essential Development Practices

- How do we design our code?
- Luckily, most of this isn't necessarily Perl specific
- Not a lot of code examples here

Design the module's interface first

- A bad interface makes a module unusable
- Think about how your module will be used
- Name things from the user's perspective
 - Not from how it's implemented
- Create examples, or even "use cases" for your interface
- Keep the examples around for your documentation!

Write tests first

- Turn your examples into tests
- Create the tests before coding
- Run the tests, which should fail
- Code (correctly!)
- Now the tests should succeed
- Tests are also documentation
- So, comment your tests

Create POD templates

- Understand the standard POD
- Enhance it with local standards
- Don't keep retyping your name and email
- Consider `Module::Starter`

Use revision control

- Essential for team programming
- But even good for one person
- Great for sharing updates
- Helpful for “when did it break”
- And more importantly “who broke it”
- Also prevents potential loss from hardware failures and human failures

Use consistent APIs for arguments and config

- Require a flag in front of every arg, except filenames
- Always allow "-" as a filename
- Always use "--" to indicate "end of args"
- Consider Config::General or Config::Std
- Allow user changes to be rewritten to the configuration file for the next run

Use a consistent layout

- Pick a style, and stick with it
- Use perltidy to enforce it
- But don't waste time reformatting everything
- Life's too short to get into fights on this

Code in commented paragraphs

- Whitespace between chunks is helpful
- Chunk according to logical steps
- Interpreting @_ separated from the next step in a subroutine
- Add leading comments in front of the paragraph for a logical-step description
- Don't use comments for user docs, use POD

Throw exceptions instead of funny returns

- People might forget to test for “undef”
- But they have to work pretty hard to ignore an exception
- Exceptions can be easily caught with `eval {}`
- Exceptions can be structured with `Exception::Class`

Add new test cases before debugging

- A bug report is a test case!
- Add it to your test suite
- (You do have a test suite, right?)
- Then debug.
- You'll know the bug is gone when the test passes
- And you'll never accidentally reintroduce that bug!

Don't optimize until you benchmark

- Get the code right first
- Then make it go faster, if necessary
- Use the tools
- Benchmark
- `Devel::DProf`
- `Devel::SmallProf`

Essential Coding Practices

- Now that we've seen development...
- Let's go on to coding...

Use strict

- Barewords in the wrong place:
my @months = (jan, feb, mar, apr, may, jun,
jul, aug, sep, oct, nov, dec);
- Variables that are probably typos:
my \$bamm_bamm = 3; \$bammmbamm;
- Evil symbolic references:
\$data{fred} = "flintstone"; ...
\$data{fred}{age} = 30;
- You just set \$flintstone to 30!

Use warnings

- “use warnings” catches most beginner mistakes
- Beware “-w” on the command line
- Unless you want to debug someone else’s mistakes when you use a module
- Reduce warnings for troublesome spots:
 - { no warnings ‘redefine’; ... }

Use grammar-based identifiers

- Packages: Noun :: Adjective :: Adjective
 - Disk::DVD::Rewritable
- Variables: adjective_adjective_noun
 - \$estimated_net_worth
- Lookup vars: adjective_noun_preposition
 - %blocks_of, @sales_for
- Subs: imperative_adjective_noun[_prep]
 - get_next_cookie, eat_cake
 - get_cookie_of_type, eat_cake_using

Use lexicals, not package variables

- Lexical access is guaranteed to be in the same file
- Unless some code passes around a reference to it
- Package variables can be accessed anywhere
- Including well-intentioned but broken code

Label loops used with last/next/redo

- last/next/redo provide a limited “goto”
- But it’s still sometimes confusing
- Always label your loops for these:
 LINE: while (<>) { ... next LINE if \$cond; ... }
- The labels should be the noun of the thing being processed
- Makes “last LINE” read like English!

Don't use bareword filehandles

- Use indirect filehandles (in modern Perl):
 - `open my $foo, "<", $someinput or die;`
- They close automatically
- Can be passed to/from subroutines
- Can be stored in aggregates
- Might need `readline()` or `print {...}` though
 - `my $line = readline $inputs[3];`
 - `print { $output_for{$day} } @list;`

Unpack @_ into named variables

- `$_[3]` is just ugly
- Even compared to the rest of Perl
- Unpack your args:
 - `my ($name, $rank, $serial) = @_;`
- Use "shift" form for more breathing room:
 - `my $name = shift; # "Flintstone, Fred"`
 - `my $rank = shift; # 1..10`
 - `my $serial = shift; # 7-digit integer`

Use named parameters if more than three

- Three positional parameters is enough
- For more than three, use a hash:
 - `my %options = %{+shift};`
 - `my $name = $options{name} || die "?";`
- Pass them as a hashref:
 - `my_routine({name => "Randal"})`
- This catches the odd-number of parameters at the caller, not the callee.

Always use explicit return

- Perl returns the last expression evaluated
- But make it explicit:
 - `return $this;`
- Easier to see that the return value is expected
- Protects against someone adding an extra line to the subroutine, breaking the return
- Also use `"return;"` for undef/empty list return, not `"return undef;"`

Use /xms and \A and \z

- /x permits whitespace and comments
- /m says “^” and “\$” match embedded newlines
- /s means “.” really matches newline
- \A is the old ^
- \z is the old \$
 - Except that it’s really “end of string”
 - Not “end of string but maybe before \n”

Use capturing parens only when capturing

- Every (...) in a regex is a capture regex
- This affects \$1, etc
- But it also affects speed
- When you don't need to capture, don't!
- Use (?: ...)
- Yes, a little more typing
- But it's clear that you don't need that value

Name your captures

- There can be quite a distance between
 - `/(w+) \s+ (\w+)/xms`
 - `$1, $2`
- Easy to make mistake, or modify and break things
- Instead, name your captures when you can:
 - `my ($given, $surname) = /(\w+)\s+(\w+)/;`
- If this match fails in a scalar context, it returns false

Export subroutines, not data

- Yes, let's create a module, but then reintroduce global variables
- OK, that's a bad idea
- Variables have only get/set interface
- And very little checking (unless you "tie")
- Export subroutines to give more control
- And more flexibility for the future

Essential Module Practices

- Once you get above 100 lines or so, modules are essential
- The unit of reuse of coding
- Modules also generally involve other people
- So it's good to get it right
- Let's look at some practices...

Use Test::More

- Testing testing testing!
- Don't code your tests ad-hoc
- Use the proven Perl framework
- "perldoc Test::Tutorial"
- For local things, create additional Test::*
mixins

Create constants with the Readonly module

- Don't use "constant", even though it's core
 - use constant PI => 3;
- You can't interpolate these easily:
 - print "In indiana, pi is @{\$PI}\n";
- Instead, get Readonly from the CPAN:
 - Readonly my \$PI => 3;
- Now you can interpolate:
 - print "In indiana, pi is \$PI\n";
- Yeay

Use the semi-builtins in Scalar::Util and List::Util

- These are core with modern Perl
- Scalar::Util has: blessed, refaddr, reftype, readonly, tainted, openhandle, weaken, is_weak, looks_like_number
- List::Util has: first, max, maxstr, min, minstr, shuffle, sum, reduce
- reduce is cool:
 - `my $factorial = reduce { $a * $b } 1..20;`
 - `my $commy = reduce { "$a, $b" } @strings;`

Use IO::Prompt for prompting

- Get it from the CPAN
- `my $line = prompt "line, please? ";`
- Automatically chomps (yeay!)
- `my $password = prompt "password:",
-echo => '*';`
- `my $choice = prompt 'letter', -onechar,
-require => { 'must be [a-e]' => qr/[a-e]/ };`
- Many more options

Use Carp and Exception::Class

- Carp blames someone else
 - use Carp qw(croak);
... open my \$f, \$file or croak "\$file?";
- Use Exception::Class for nice exceptions
- Exception::Class-based exceptions capture caller and current filename/linenumber/package
- They also stringify nicely for naive displays
- Create hierarchies for classified handling

Use Fatal to turn failures into exceptions

- Tired of forgetting "or die" on "open"?
- use Fatal qw(:void open);
- ...
- open my \$f, "BOGUS FILE"; # dies!
- Yes, finally open does the right thing
- Autodetects void context, so this still works:
 - unless (open my \$f, "BOGUS") { ... }

Use Data::Alias or Lexical::Alias

- Don't subscript more than once in a loop:
 - for my \$k (sort keys %names) {
 if (is_bad(\$names{\$k})) {
 print "\$names{\$k} is bad"; ...
- Instead, alias the value:
 - for my \$k (sort keys %names) {
 alias my \$name = \$names[\$k];
- The \$name is read/write!
- Requires Data::Alias or Lexical::Alias

Use Regexp::Common

- Don't reinvent the common regex
- Use the expert coding in Regexp::Common
- `$number =~ /$RE{num}{real}/`
- `$balanced_parens =~ /$RE{balanced}/`
- `$bad_words =~ /$RE{profanity}/`
- Module comes with 175,000 tests!

Use Class::Std or Object::InsideOut

- Getting accessors and privacy correct can take special care
- Use inside-out objects for safety and best development speed
- Class::Std or Object::InsideOut

Use Perl::Critic

- Automatically test your code against various suggestions made here
- Can be configured to require or ignore guidelines
- Consider it “lint for Perl”

In summary

- Learn from the ones who have the most scars
- Buy the book...
- Damian could use the money!

